

Section Handout #4—Data Structures, Encryption

Problem 1: Roman Numerals

In Roman numerals, characters of the alphabet are used to represent integers as shown in this table:

symbol	value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Each character in a Roman numeral stands for the corresponding value. Ordinarily, the value of the Roman numeral as a whole is the sum of the individual character values given in the table. Thus, the string "**LXXVI**" denotes $50 + 10 + 10 + 5 + 1$, or 76. The only exception occurs when a character corresponding to a smaller value precedes a character representing a larger one, in which case the value of the first letter is subtracted from the total, so that the string "**IX**" corresponds to $10 - 1$, or 9.

Write a function **romanToDecimal** that takes a string representing a Roman numeral alongside a map associating symbols (e.g., "**L**") to decimal values (e.g., 50) and returns the corresponding decimal number. To find the values of each Roman numeral character, your function should look that character up in a map that implements the Roman numeral conversion. If the string contains characters that are not in the table, **romanToDecimal** should return -1 .

```
function romanToDecimal(numeral, symbolToValueMap) {
```

Problem 2: Scaling Recipes

I've decided to throw the biggest of dinner parties this coming weekend, and I've further decided to cook everything myself. I know what I'm cooking, and I've even converged on the collection of recipes I'll follow. Because you're invited, you're offering to write a JavaScript program to scale up recipes so I know precisely how much of each ingredient to buy at the supermarket.

Take for example this lovely soup dish I found online, coincidentally expressed in JSON:

```
let soup = {
  name: "Charred Cauliflower Stew",
  servings: 4,
  ingredients: [
    { amount: 2, unit: "head", name: "cauliflower"},
    ... many more ingredients omitted
    { amount: 1, unit: "cup", name: "watercress"},
  ],
  instructions: [
    "Heat the grill to high and lightly oil the grates.",
    ... many more instructions omitted
    "Garnish with remaining watercress and serve."
  ]
};
```

The problem? If I need to cook for 16 people instead of 4, I'd prefer the recipe be structured as follows:

```
{
  name: "Charred Cauliflower Stew",
  servings: 16,
  ingredients: [
    { amount: 8, unit: "head", name: "cauliflower"},
    ... many more ingredients omitted
    { amount: 1, unit: "quart", name: "watercress"},
  ],
  instructions: [
    "Heat the grill to high and lightly oil the grates.",
    ... many more instructions omitted
    "Garnish with remaining watercress and serve."
  ]
};
```

Since the recipe was quadrupled, 2 teaspoons became 8 teaspoons and 1 cup became 4 cups. But those quantities are better expressed in larger units of measure, which is why 8 teaspoons isn't 8 teaspoons or even 2.66 tablespoons, but rather 1.33 ounces. When a quantity can be expressed in terms of 1.0 or more of the next larger unit of measure, it is. If instead of a party of 16 I throw a party for 1600 (**#lifegoals**), I'll need a drastically reformed ingredient list for the soup, as with:

```

{
  name: "Charred Cauliflower Stew",
  servings: 1600,
  ingredients: [
    { amount: 800, unit: "head", name: "cauliflower"},
    ... many more ingredients omitted
    { amount: 3.125, unit: "bushel", name: "watercress"},
  ],
  instructions: [
    "Heat the grill to high and lightly oil the grates.",
    ... many more instructions omitted
    "Garnish with remaining watercress and serve."
  ]
};

```

These conversions are possible, provided we have an ordered list of conversions for each of the standard units of measures. Fortunately, we do, because the internet has everything.

```

const CONVERSIONS = [
  { unit: "dram", amount: 4/3 },
  { unit: "teaspoon", amount: 3 },
  { unit: "tablespoon", amount: 2 },
  { unit: "ounce", amount: 8 },
  { unit: "cup", amount: 2 },
  { unit: "pint", amount: 2 },
  { unit: "quart", amount: 4 },
  { unit: "gallon", amount: 2 },
  { unit: "peck", amount: 4 },
  { unit: "bushel", amount: 55/7 },
  { unit: "barrel", amount: 6000},
  { unit: "acre" } // no larger unit of measure, so no amount field
];

```

Each object in the array constant stores the number of a particular unit needed to make up exactly one of the next larger unit. Therefore, 3 teaspoons make up one tablespoon, 2 tablespoons make up one ounce, 8 ounces make up one cup, and so forth.

Your job as a section it to implement the **scale** function, which accepts a recipe object (structured as those you've seen above) and the desired number of servings, and modifies the recipe object the house the amounts needed to serve those many people. If an ingredient's unit of measure appears in **CONVERSIONS**, the quantity and unit of measure should be normalized as they have been in the above examples. You must allow for the possibility the supplied recipe needs to be scaled up or down. Hint: convert **all** quantities to drams and then normalize up to the most sensible unit of measure.

```
const CONVERSIONS = [
  { unit: "dram", amount: 4/3 }, { unit: "teaspoon", amount: 3 },
  { unit: "tablespoon", amount: 2 }, { unit: "ounce", amount: 8 },
  { unit: "cup", amount: 2 }, { unit: "pint", amount: 2 },
  { unit: "quart", amount: 4 }, { unit: "gallon", amount: 2 },
  { unit: "peck", amount: 4 }, { unit: "bushel", amount: 55/7 },
  { unit: "barrel", amount: 6000}, { unit: "acre" }
];

function scale(recipe, servings) {
```

Problem 3: Enigma Reflector Construction

For this problem, you’ll leverage your understanding of strings to generate a random reflector that you could, in theory, use in your implementation of Assignment 4.

Recall that the reflector encrypts each letter of the alphabet to some other letter and that all reflector encryptions are reversible. Thus, if A is encoded as P, then P must be encoded as an A.

The permutation for the reflector used in the Assignment 4 specification is:

"IXUHFZDAOMTKQJWNSRLCYPBVG"

Note that A is encoded as I, B is encoded as X, C is encoded as U, and so forth. And because the permutation is a reflector, it must be the case that I maps back to A, X maps back to B, U maps back to C, and so on. No character in the above permutation occupies its normal spot in the alphabet, since that would imply that character is its own encryption, and that’s not permitted.

For this problem, we want you to implement the **buildReflector** function, which constructs and returns a valid reflector permutation. Because this function is algorithmically complex, we give you the general algorithm in pseudocode:

```
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const STARTER = "-----";

function buildReflector() {
  let reflector = STARTER;
  let candidates = ALPHABET;
  while (candidates.length > 0) {
    select a random position from the string of candidates, and remember the character there
    remove the selected character from the string of candidates
    select a random position from the updated string, and remember the character there
    remove the selected character from the string of candidates
    update the reflector so that each of the two selected characters occupies the other's
    spot in the alphabet
  }
  return reflector;
}
```

Note that two characters are deleted from **candidates** on each iteration, which means that the body of the loop will always execute 13 times. And recall the **randomInteger** function from Assignment 1, where a call to **randomInteger(a,b)** for integers **a** and **b** is equally likely to return any integer between **a** and **b**, inclusive.

```
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const STARTER = "-----";

function buildReflector() {
  let reflector = STARTER;
  let candidates = ALPHABET;
  // flesh out the rest of your implementation here
}
```

Problem 4: Move-To-Front Encryption

Some encryption schemes rely on ever-evolving encryption keys to more effectively disguise common letters like 'e', 't', and 'a'. For this problem, you'll write a function that encrypts a lowercase string using a **move-to-front encryption key transformation**.

The key is first set to be the lowercase alphabet, "abcdefghijklmnopqrstuvwxyz". For each character **ch** in the original word, the encryption scheme:

- finds the index of **ch** within the key,
- appends the character at that index within the alphabet to a running encryption, and
- moves **ch** to the front of the key while keeping the other letters in the same order.

As an example, the word "ooze" would be transformed into "oazg", and here's why:

- Start with the key of "abcdefghijklmnopqrstuvwxyz".
- The first letter is 'o' and appears at index 14 of the key.
 - Append the 14th letter of the alphabet—that is, an 'o'—to the encryption.
 - Move 'o' to the front of the key to yield "oabcdefghijklmnopqrstuvwxyz".
- The second letter is another 'o', but it now appears at index 0 of the key.
 - Append the 0th letter of the alphabet, 'a', to the encryption.
 - The key remains the same.
- The third letter 'z' appears at index 25 of the key.
 - Append the 25th letter of the alphabet, 'z', to the encryption.
 - Move 'z' to the front to yield "zoabcdefghijklmnopqrstuvwxyz".
- The final letter 'e' appears at index 6 of the key.
 - Append the 6th letter of the alphabet, 'g', to the encryption.
 - Move 'e' to the front, producing "ezoabcdefghijklmnopqrstuvwxyz".

The final encryption? "oazg"! That's what would be returned.

Your job? Implement the **encrypt** function to encrypt the supplied word—assumed to consist of lowercase letters and nothing else—and returns its encryption.

```
const ALPHABET = "abcdefghijklmnopqrstuvwxyz"
function encrypt(word) {
  // present your implementation of encrypt right here
}
```